Semantic Search: Reconciling Expressive Querying and Exploratory Search

Sébastien Ferré, Alice Hermann

Semantic Search: Reconciling Expressive Querying and Exploratory Search

Sébastien Ferré[1] and Alice Hermann[2]

[1] IRISA/Université de Rennes 1, Campus de Beaulieu, 35042 Rennes cedex, France
ferre@irisa.fr
[2] IRISA/INSA de Rennes, Campus de Beaulieu, 35708 Rennes cedex 7, France
alice.hermann@irisa.fr

Abstract. Faceted search and querying are two well-known paradigms to search the Semantic Web. Querying languages, such as SPARQL, offer expressive means for searching RDF datasets, but they are difficult to use. Query assistants help users to write well-formed queries, but they do not prevent empty results. Faceted search supports exploratory search, i.e., guided navigation that returns rich feedbacks to users, and prevents them to fall in dead-ends (empty results). However, faceted search systems do not offer the same expressiveness as query languages. We introduce *Query-based Faceted Search* (QFS), the combination of an expressive query language and faceted search, to reconcile the two paradigms. In this paper, the LISQL query language generalizes existing semantic faceted search systems, and covers most features of SPARQL. A prototype, Sewelis (aka. Camelis 2), has been implemented, and a usability evaluation demonstrated that QFS retains the ease-of-use of faceted search, and enables users to build complex queries with little training.

1 Introduction

With the growing amount of available resources in the Semantic Web (SW), it is a key issue to provide an easy and effective access to them, not only to specialists, but also to casual users. The challenge is not only to allow users to retrieve particular resources (e.g., flights), but to support them in the exploration of a knowledge base (e.g., which are the destinations? Which are the most frequent flights? With which companies and at which price?). We call the first mode *retrieval search*, and, following Marchionini [10], the second mode *exploratory search*. Exploratory search is often associated to *faceted search* [5,13], but it is also at the core of Logical Information Systems (LIS) [4,2], and Dynamic Taxonomies [12]. Exploratory search allows users to find information without *a priori* knowledge about either the data or its schema. Faceted search works by suggesting restrictions, i.e., selectors for subsets of the current selection of items. Restrictions are organized into facets, and only those that share items with the current selection are suggested. This has the advantage to provide guided navigation, and to prevent dead-ends, i.e., empty selections. Therefore, faceted search is *easy-to-use* and *safe*: *easy-to-use* because users only have to

choose among the suggested restrictions, and *safe* because, whatever the choice made by users, the resulting selection is not empty. The selections that can be reached by navigation correspond to queries that are generally limited to conjunctions of restrictions, possibly with negation and disjunction on values. This is far from the expressiveness of query languages for the semantic web, such as SPARQL[3]. There are *semantic faceted search* that extend the expressiveness of reachable queries, but still to a small fragment of SPARQL (e.g., SlashFacet [7], BrowseRDF [11], SOR [9], gFacet [6]). For instance, none of them allow for cycles in graph patterns, unions of complex graph patterns, or negations of complex graph patterns.

Querying languages for the semantic web are quite expressive but are difficult to use, even for specialists. Users are asked to fill an empty field (problem of the writer's block), and nothing prevents them to write a query that has no answer (dead-end). Even if users have a perfect knowledge of the syntax and semantics of the query language, they may be ignorant about the data schema, i.e., the *ontology*. If they also master the ontology or if they use a graphical query editor (e.g., SemanticCrystal [8], SCRIBO Graphical Editor[4]) or an auto-completion system (e.g., Ginseng [8]) or keyword query translation (e.g., Hermes [14]), the query will be syntactically correct and semantically consistent w.r.t. the ontology but it can still produce no answer.

The contribution of this paper, *Query-based Faceted Search* (QFS), is to define a semantic search that is (1) easy to use, (2) safe, and (3) expressive. Ease-of-use and safeness are retained from existing faceted search systems by keeping their general principles, as well as the visual aspect of their interface. Expressiveness is obtained by representing the current selection by a *query* rather than by a set of items, and by representing navigation links by *query transformations* rather than by set operations (e.g., intersection, crossing). In this way, the expressiveness of faceted search is determined by the expressiveness of the query language, rather than by the combinatorics of user interface controls. In this paper, the query language, named LISQL, generalizes existing semantic faceted search systems, and covers most features of SPARQL. The use of queries for representing selections in faceted search has other benefits than navigation expressiveness. The current query is an intensional description of the current selection that complements its extensional description (list of items). It informs users in a precise and concise way about their exact position in the navigation space. It can easily be copied and pasted, stored and retrieved later. Finally, it allows expert users to modify the query by hand at any stage of the navigation process, without loosing the ability to proceed by navigation.

The paper is organized as follows. Section 2 discusses the limits of set-based faceted search by formalizing the navigation from selection to selection. Section 3 introduces LISQL queries and their transformations. In Section 4, navigation with QFS is formalized and proved to be *safe* and *complete* w.r.t. LISQL, and *efficient*. Section 5 reports about a usability evaluation, and Section 6 concludes.

[3] see `http://www.w3.org/TR/rdf-sparql-query/`

[4] `http://www.scribo.ws/xwiki/bin/view/Blog/SparqlGraphicalEditor`

2 Limits of Set-based Faceted Search

The principle of faceted search [13] is to guide users from *selection* of items to selection of items. At each navigation step, a new selection is derived by applying a set operation between the current selection S and a *restriction* R. A restriction is a *feature* that applies to at least one item of the current selection, i.e., $S \cap R \neq \emptyset$. Typically, a feature is a pair facet-value, and the set operation is intersection: $S := S \cap R$. The new selection is the set of items that belong to the current selection, and that belong to the restriction. Extensions of faceted search may allow for the exclusion of a restriction ($S := S \setminus R$), or the union with a restriction ($S := S \cup R$). Restrictions can also be tags or item names.

In the context of the Semantic Web, items and values are resources, facets are properties, and tags are classes. Because of the relational nature of semantic data, new kinds of restrictions and set operations have been introduced in semantic faceted search (e.g., /facet [7], BrowseRDF [11], SOR [9], gFacet [6]). A restriction can be the set of items that are subject of some property (the domain of the property), or that are object of some property (the range of the property) (e.g., BrowseRDF). A facet can be defined as a path of properties. Finally, a property p can be crossed forwards ($S := p(S, .)$) or backwards ($S := p(., S)$) (e.g., /facet, SOR, gFacet).

Both in theory and in practice, it is useful to distinguish between syntax and semantics. For example, we should distinguish between a pair facet-value (syntax), and the set of items it matches (semantics). In the following table, we define the syntax and semantics of the various kinds of restrictions: r denotes any RDF resource (URI, literal), c denotes a RDFS class, p denotes a RDF property, and S_0 denotes the set of all items (possibly all resources of a RDF dataset).

restriction	syntax	semantics	examples
name	r	$\{r\}$	`<JohnSmith>`, `"John"`, `2011`
tag	`a` c	rdf:type$(., \{c\})$	`a person`
(facet, value)	p `:` r	$p(., \{r\})$	`year : 2011`
(facet, value)	p `of` r	$p(\{r\}, .)$	`mother of <JohnSmith>`
domain	p `: ?`	$p(., S_0)$	`year : ?`
range	p `of ?`	$p(S_0, .)$	`mother of ?`

The same distinction can be made for complex selections, and we introduce in the following table a syntax for the various set operations that can be applied between selections and restrictions: S denotes a selection, and R denotes a restriction that is relevant to S: i.e., $S \cap R \neq \emptyset$.

selection	syntax	semantics
initial	`?`	S_0
intersection	S `and` R	$S \cap R$
exclusion	S `and not` R	$S \setminus R$
union	S `or` R	$S \cup R$
crossing backwards	p `:` S	$p(., S)$
crossing forwards	p `of` S	$p(S, .)$

The syntactic form of restrictions are *features*. The syntactic form of selections are *queries* whose answers are sets of items, i.e., subsets of S_0. The above tables implicitly define a grammar for features and queries:

$$S \rightarrow \texttt{?} \mid S \texttt{ and } R \mid S \texttt{ and not } R \mid S \texttt{ or } R \mid p \texttt{ : } S \mid p \texttt{ of } S$$
$$R \rightarrow r \mid \texttt{a } c \mid p \texttt{ : } r \mid p \texttt{ of } r \mid p \texttt{ : ?} \mid p \texttt{ of ?}.$$

This grammar already defines a rich language of accessible queries, but it has strong limits in terms of flexibility and expressivity, as we discuss now. To reach some selections requires a precise ordering in navigation steps, which hinders the flexibility of the search, and assumes that the user has a clear idea of his query in advance. For example, to reach the query `father of (mother of (name : "John") and name : "Jane")`, the user has first to select `name : "John"` (*people named John*), then to cross forward `mother` (*their mothers*), then to intersect with `name : "Jane"` (*... whose name is Jane*), and finally to cross forward `father` (*their fathers*). Any other ordering will fail; starting from the expected result (grand-fathers) will lead to the set of grand-children instead.

Some useful selections that can be defined in terms of set operations are not reachable by set-based faceted search. For example, the following kinds of selections are not reachable: unions of complex selections. e.g., $(R_1 \cap R_2) \cup (R_3 \cap R_4)$; or intersection of crossings from complex selections, e.g., $p_1(., R_1 \cap R_2) \cap p_2(., R_3 \cap R_4)$. Note that a selection $S_1 \cap p(., S_2)$ cannot in general be obtained by first navigating to S_1, then crossing forwards p, navigating to S_2, and finally crossing backwards p, because it is not equivalent to $p(., p(S_1, .) \cap S_2)$ unless p is inverse functional. Therefore, not all combinations of intersection, union, and crossing are reachable, which is counter-intuitive and limiting for end users.

Existing approaches to semantic faceted search often have additional limitations, which are sometimes hidden behind a lack of formalization. A same facet (a property path) cannot be used several times, which is fine for functional properties but not for relations such as "child": $p(., f_1 \cap f_2)$ is reachable but not $p(., f_1) \cap p(., f_2)$ (e.g., BrowseRDF, gFacet). A property whose domain and range are the same cannot be used as a facet (e.g., /facet), which includes all family and friend relationships for instance.

3 Expressive Queries and their Transformations

The contribution of our approach, *Query-based Faceted Search* (QFS), is to significantly improve the expressivity of faceted search, while retaining its properties of safeness (no dead-end), and ease-of-use. The key idea is to define navigation steps at the syntactic level as query transformations, rather than at the semantic level as set operations. The navigation from selection to selection, as well as the computation of restrictions related to the current selection, are retained by defining the semantics of features and queries, i.e., the mapping from a feature f or a query q to a set of items: $R = \mathit{items}(f)$ and $S = \mathit{items}(q)$. Transformations at the syntactic level are necessary because there exist useful navigation steps that cannot be obtained by applying set operations on the current selection. For

example, given $S = R_1 \cap R_2$, the set of items $S' = R_1 \cap (R_2 \cup R_3)$ cannot be derived from S and R_3. On the contrary, the query f_1 `and (`f_2 `or` f_3`)` can be derived from the query f_1 `and` f_2 and the feature f_3 because enough information is retained at the syntactic level.

In this section, we generalize in a natural way the set of queries compared to Section 2. This defines a query language, which we call LISQL (LIS Query Language). We then define a set of query transformations so that every LISQL query can be reached in a finite sequence of such transformations. This is in contrast with previous contributions in faceted search that introduce new selection transformations, and leave the query language implicit. We think that making the language of reachable queries explicit is important for reasoning on and comparing different faceted search systems. In Section 3.3, we give a translation from LISQL to SPARQL, the reference query language of the Semantic Web. This provides both a way to compute the answers of queries with existing tools, and a way to evaluate the level of expressivity achieved by LISQL.

3.1 The LIS Query Language (LISQL)

A more general query language, LISQL, can be obtained simply by merging the syntactic categories of features and queries in the grammar of Section 2, so that every query can be used in place of a feature.

Definition 1 (LISQL queries). *The syntax and semantics of the LISQL constructs is defined in the following table, where r is a resource, c is a class, p is a property, S_0 is the set of all items, and q_1, q_2 are LISQL queries s.t. $S_1 = items(q_1)$ and $S_2 = items(q_2)$.*

query	*syntax (q)*	*semantics (items(q))*
resource	r	$\{r\}$
class	`a` c	*rdf:type*$(., \{c\})$
all	`?`	S_0
crossing backwards	p `:` q_1	$p(., S_1)$
crossing forwards	p `of` q_1	$p(S_1, .)$
complement	`not` q_1	$S_0 \setminus S_1$
intersection	q_1 `and` q_2	$S_1 \cap S_2$
union	q_1 `or` q_2	$S_1 \cup S_2$

The definition of LISQL allows for the arbitrary combination of intersection, union, complement, and crossings. In order to further improve the expressiveness of LISQL from tree patterns to graph patterns, we add variables (e.g., `?X`) as an additional construct. Variables serve as co-references between distant parts of the query, and allows for the expression of cycles. For example, the query that selects people who are an employee of their own father can be expressed as `a person and father : ?X and employee of ?X`, or alternately as `a person and ?X and employee of father of ?X`. The semantics of queries with variables is given with the translation to SPARQL in Section 3.3, because it cannot be defined like in the table of Definition 1.

Syntactic constructs are given in increasing priority order, and brackets are used in concrete syntax for disambiguation. The most general query `?` is a neutral element for intersection, and an absorbing element for union. In the following, we use the example query q_{ex} = `a person and birth : (year : (1601 or 1649) and place : (?X and part of England)) and father : birth : place : not ?X`, which uses all constructs of LISQL, and selects the set of "persons born in 1601 or 1649 at some place in England, and whose father is born at another place".

3.2 Query Transformations

We have generalized the query language by allowing complex selections in place of restrictions: e.g., $S_1 \cap S_2$ instead of $S \cap R$. However, because the number of suggested restrictions in faceted search must be finite, it is not possible to suggest arbitrarily complex restrictions. More precisely, the *vocabulary* of features must be finite. In QFS, we retain the same set of features as in Section 2, which is a finite subset of LISQL for any given dataset.

The key notion we introduce to reconcile this finite vocabulary, and the reachability of arbitrary LISQL queries is the notion of *focus* in a query.

Definition 2 (focus). *A* focus *of a LISQL query q is a node of the syntax tree of q, or equivalently, a subquery of q. The set of foci of q is noted $\Phi(q)$; the* root focus *corresponds to the root of the syntax tree, and represents the whole query. The subquery at focus $\phi \in \Phi(q)$ is noted $q[\phi]$; and $q[\phi := q_1]$ denotes the modified query q, where the subquery at focus ϕ has been replaced by q_1.*

In the following, when it is necessary to refer to a focus in a query, the corresponding subquery is underlined with the focus name as a subscript, like in `mother of` $\underline{?}_\phi$. Foci are used in QFS to specify on which subquery a query transformation should be applied. For example, the query (`(`f_1 `and` f_2`) or (`f_3 `and` f_4`)` can be reached from the query (`(`f_1 `and` f_2`) or` f_3 by applying the intersection with restriction f_4 to the subquery f_3, instead of to the whole query. Similarly, the query p_1 `: (`f_1 `and` f_2`) and` p_2 `: (`f_3 `and` f_4`)` can be reached by applying the intersection with restriction f_4 to the subquery f_3. This removes the problem of unreachable selections in set-based faceted search presented in Section 2. Moreover, this removes the need for a strategy in the ordering of navigation steps. For example, the query `a woman and mother of name : "John"` can be reached by first selecting `a woman`, then by selecting `mother of` $\underline{?}_\phi$, then by inserting `name : "John"` at the focus ϕ.

Definition 3 (query transformation). *The different kinds of LISQL query transformations are listed in the following table, where each transformation is paramaterized by a focus ϕ and a query q_1. The expression $q[t]$ is the query that results from the application of transformation t to query q.*

transformation	*notation (t)*	*result query (q[t])*
intersection	ϕ `and` q_1	$q[\phi :=$ $q[\phi]$ `and` $q_1]$
exclusion	ϕ `and not` q_1	$q[\phi :=$ $q[\phi]$ `and not` $q_1]$
union	ϕ `or` q_1	$q[\phi :=$ $q[\phi]$ `or` $q_1]$

We show in the following equations how the intersection with an arbitrary LISQL query can be recursively decomposed into a finite sequence of intersections with features, and exclusions and unions with the most general query `?`.

$$
\begin{aligned}
q[\phi \texttt{ and (?)}] &= q \\
q[\phi \texttt{ and (}p \texttt{ : } q_1\texttt{)}] &= q[\phi \texttt{ and } p \texttt{ : } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1] \\
q[\phi \texttt{ and (}p \texttt{ of } q_1\texttt{)}] &= q[\phi \texttt{ and } p \texttt{ of } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1] \\
q[\phi \texttt{ and (not } q_1\texttt{)}] &= q[\phi \texttt{ and not } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1] \\
q[\phi \texttt{ and (}q_1 \texttt{ and } q_2\texttt{)}] &= q[\phi \texttt{ and } \underline{q_1}_{\phi_1}][\phi_1 \texttt{ and } q_2] \\
q[\phi \texttt{ and (}q_1 \texttt{ or } q_2\texttt{)}] &= q[\phi \texttt{ and } \underline{q_1}_{\phi_1}][\phi_1 \texttt{ or } \underline{?}_{\phi_2}][\phi_2 \texttt{ and } q_2]
\end{aligned}
$$

For example, the complex query q_{ex} = `a person and birth : (year : (1601 or 1649) and place : (?X and part of England)) and father : birth : place : not ?X` can be reached through the navigation path: $\underline{?}_{\phi_0}[\phi_0$ `and a person`$]$ $[\phi_0$ `and birth :` $\underline{?}_{\phi_1}]$ $[\phi_1$ `and year :` $\underline{1601}_{\phi_2}]$ $[\phi_2$ `or` $\underline{?}_{\phi_3}]$ $[\phi_3$ `and 1649`$]$ $[\phi_1$ `and place :` $\underline{?}_{\phi_4}]$ $[\phi_4$ `and ?X`$]$ $[\phi_4$ `and part of England`$]$ $[\phi_0$ `and father :` $\underline{?}_{\phi_5}]$ $[\phi_5$ `and birth :` $\underline{?}_{\phi_6}]$ $[\phi_6$ `and place :` $\underline{?}_{\phi_7}]$ $[\phi_7$ `and not` $\underline{?}_{\phi_8}]$ $[\phi_8$ `and ?X`$]$. The classical facet-value features appear to be redundant for navigation as their intersection can be decomposed, but they are still useful for visualization in a faceted search interface.

Sequences of query transformations are analogous to the use of graphical query editors, but the key difference is that answers and restrictions are returned at each step, providing feedback, understanding-at-a-glance, no dead-end, and all benefits of exploratory search. Despite the syntax-based definition of navigation steps, those have a clear semantic counterpart. Intersection is the same as in standard faceted search, only making it available on the different entities involved in the current query. In the above example, intersection is alternately applied to the person, his birth, his birth's place, his father, etc. The set of relevant restrictions is obviously different at different foci. The union transformation introduces an alternative to some subquery (e.g., an alternative birth's year). The exclusion transformation introduces a set of exceptions to the subquery (e.g., excluding some father's birth's place). In Section 4, we precisely define which query transformations are suggested at each navigation step, and we prove that the resulting navigation graph is safe (no dead-end), and complete (every "safe" query is reachable).

3.3 Translation to and Comparison with SPARQL

We here propose a (naive) translation of LISQL queries to SPARQL queries. It involves the introduction of variables that are implicit in LISQL queries. As this translation applies to LISQL queries with co-reference variables, it becomes possible to compute their set of items.

Definition 4 (SPARQL translation). *The* SPARQL translation *of a LISQL query q is $sparql(q)$ =* `SELECT DISTINCT` $?x$ `WHERE` $\{\ S_0(x)\ \ GP(x,q)\ \}$*, where*

the graph pattern $S_0(x)$ binds x to any element of the set of all items S_0, and the function GP inductively defines the graph pattern of q with variable x representing the root focus.

$$
\begin{array}{ll}
GP(x, \texttt{?}v) & = S_0(v)\ \texttt{FILTER (?}x\ \texttt{= ?}v\texttt{)} \\
GP(x, r) & = \texttt{FILTER (?}x\ \texttt{=}\ r\texttt{)} \\
GP(x, \texttt{a}\ c) & = \texttt{?}x\ \texttt{rdf:type}\ c \\
GP(x, p\ \texttt{:}\ q_1) & = \texttt{?}x\ p\ \texttt{?}y\texttt{.}\ GP(y, q_1) \quad \textit{where } y \textit{ is a fresh variable} \\
GP(x, p\ \texttt{of}\ q_1) & = \texttt{?}y\ p\ \texttt{?}x\texttt{.}\ GP(y, q_1) \quad \textit{where } y \textit{ is a fresh variable} \\
GP(x, \texttt{?}) & = \{\ \} \\
GP(x, \texttt{not}\ q_1) & = \texttt{NOT EXISTS}\ \{\ GP(x, q_1)\ \} \\
GP(x, q_1\ \texttt{and}\ q_2) & = GP(x, q_1)\ GP(x, q_2) \\
GP(x, q_1\ \texttt{or}\ q_2) & = \{\ GP(x, q_1)\ \}\ \texttt{UNION}\ \{\ GP(x, q_2)\ \}
\end{array}
$$

We now discuss the translations of LISQL queries compared to SPARQL in general. They have only one variable in the SELECT clause because of the nature of faceted search, i.e., navigation from set to set. From SPARQL 1.0, LISQL misses the optional graph pattern, and the named graph pattern. Optional graph patterns are mostly useful when there are several variables in the SELECT clause. LISQL has the `NOT EXISTS` construct of SPARQL 1.1. If we look at the graph patterns generated for intersection and union, the two subpatterns necessarily share at least one variable, x. This is a restriction compared to SPARQL, but one that makes little difference in practice as disconnected graph patterns are hardly useful in practice.

4 A Safe and Complete Navigation Graph

In this section, we formally define the navigation space over a RDF dataset as a graph, where vertices are navigation places, and edges are navigation links. A navigation place is made of a query q and a focus ϕ of this query. The focus determines the selection of items to be displayed, and the corresponding restrictions at this focus. A navigation link is defined by a query transformation and, possibly, a focus move. Before defining the navigation graph itself, we first define the set of items and the set of restrictions for some query q and some focus $\phi \in \Phi(q)$. The set of items is defined as the set of items of the query $\mathit{flip}(q, \phi)$ that is the reformulation of q from the point of view of the focus ϕ. For example, the reformulation, called the *flip*, of the query `a woman and mother of` $\underline{\texttt{name : "John"}}_\phi$ is the query `name : "John" and mother : a woman`.

Definition 5 (flip at focus). *The* flip *of a query q at a focus $\phi \in \Phi(q)$ is defined as $\mathit{flip}(q, \phi) = \mathit{flip}'(\texttt{?}, q, \phi)$, where the function $\mathit{flip}'(k, q', \phi)$ is inductively defined, with k representing the context of q' in q, by (only main cases are given):*

$$
\begin{array}{lll}
\mathit{flip}(k, p\ \texttt{:}\ q_1, \phi) & = \mathit{flip}(p\ \texttt{of}\ k, q_1, \phi) & \textit{if } \phi \in q_1 \\
\mathit{flip}(k, q_1\ \texttt{and}\ q_2, \phi) & = \mathit{flip}(k\ \texttt{and}\ q_2, q_1, \phi) & \textit{if } \phi \in q_1 \\
\mathit{flip}(k, q_1\ \texttt{or}\ q_2, \phi) & = \mathit{flip}(k, q_1, \phi) & \textit{if } \phi \in q_1 \\
\mathit{flip}(k, \texttt{not}\ q_1, \phi) & = \mathit{flip}(k, q_1, \phi) & \textit{if } \phi \in q_1 \\
\mathit{flip}(k, q', \phi) & = q'\ \texttt{and}\ k & \textit{otherwise}
\end{array}
$$

When the focus is in the scope of an union, only the alternative that contains the focus is used in the flipped query. This is necessary to have the correct set of restrictions at that focus, and this is also useful to access the different subselections that compose an union. For example, in the query `a man and (firstname : "John"`$_\phi$ `or lastname : "John")`, the focus ϕ allows to know the set of men whose firstname is John without forgetting the second alternative. When the focus is in the scope of a complement, this complement is ignored in the flipped query. This is useful to access the subselection to be excluded. For example, in the query `a man and not father : ?`$_\phi$, the focus ϕ allows to know the set of men who have a father, i.e., those who are to be excluded from the selection of men.

Definition 6 (items at focus). *The* items *of a query q at focus ϕ is defined as the items of the* flip *of q at focus ϕ, i.e., $items(q, \phi) = items(flip(q, \phi))$.*

This enables the definition of the set of restrictions at each focus in the normal way. The navigation graph can then be formally defined.

Definition 7 (restrictions at focus). *The* restrictions *of a query q at focus ϕ is defined as the features that share items with the query q at focus ϕ:*

$$restr(q, \phi) = \{f \mid items(q, \phi) \cap items(f) \neq \emptyset\}.$$

Definition 8 (navigation graph). *Let D be a RDF dataset. The* navigation graph *$G_D = (V, E)$ of D has its set of vertices defined by*

$$V = \{(q, \phi) \mid q \in LISQL, \phi \in \Phi(q)\},$$

and its set of edges defined by the following table for every vertice (q, ϕ). The notation $(q', \phi') = (q, \phi)[l]$ denotes the navigation place obtained by traversing the navigation link l from the navigation place (q, ϕ).

navigation link	*notation (l)*	*target ((q', ϕ'))*	*conditions*
focus change	`focus` ϕ'	(q, ϕ')	*for every focus* $\phi' \in \Phi(q)$
intersection	`and` f	$(q[\phi$ `and` $\underline{f}_{\phi'}], \phi')$	*for every* $f \in restr(q, \phi)$
exclusion	`and not ?`	$(q[\phi$ `and not` $\underline{?}_{\phi'}], \phi')$	
union	`or ?`	$(q[\phi$ `or` $\underline{?}_{\phi'}], \phi')$	
name	`name ?`v	$(q[\phi$ `and` $\underline{?v}_{\phi'}], \phi')$	*for some fresh variable* v
reference	`ref ?`v	$(q[\phi$ `and` $\underline{?v}_{\phi'}], \phi')$	*for every* $v \in vars(q)$ *s.t.* $items(q', \phi') \neq \emptyset$
delete	`delete`	$(q[\phi := ?], \phi)$	

The number of navigation places (vertices) is infinite because there are infinetely many LISQL queries, but the number of outgoing navigation links (edges) is finite at each navigation place because the vocabulary of features is finite, and the number of foci and variables in a query is finite. By default, the initial navigation place is $v_0 = (\underline{?}_\phi, \phi)$. The following lemma shows that intersection navigation links behave as in standard faceted search.

Lemma 1. *For every query q, focus $\phi \in \Phi(q)$, and feature f, the following equality holds: $items((q, \phi)[$`and` $f]) = items(q, \phi) \cap items(f)$.*

4.1 Safeness and Completeness

From the formal definition of navigation graphs, we can now formally state safeness and completeness theorems. Those theorems have subtle conditions w.r.t. focus change, and the main purpose of this section is to discuss them. For reasons of space, lemmas and proofs have been removed, but they are fully available in a research report [3] (the presentation is slightly different but equivalent).

Theorem 1 (safeness). *Let D be a RDF dataset. The navigation graph G_D is* safe *except for some focus changes, i.e., for every path of navigation links without focus change from (q, ϕ) to (q', ϕ'), $items(q, \phi) \neq \emptyset$ implies $items(q', \phi') \neq \emptyset$.*

We justify to allow for unsafe focus changes by considering the following navigation scenario. The current query has the form $q = \underline{f_1 \texttt{ or } f_2}_{\phi}$, i.e., the union of two restrictions. The feature f_3 is a restriction of q such that $items(f_2) \cap items(f_3) = \emptyset$, i.e., only items of f_1 match f_3. The intersection with f_3 leads to the query $q' = (f_1 \texttt{ or } f_2) \texttt{ and } \underline{f_3}_{\phi'}$, and a focus change on f_2 leads to an empty selection. We could prevent intersection with f_3 but this would be counter-intuitive because it is a valid restriction for (q, ϕ). We could simplify the query q' by removing the second alternative f_2 ($q' = f_1 \texttt{ and } f_3$), or forbid the focus change, but we think users should have full control on the query they have built. Finally, allowing for the unsafe focus change is a simple way to inform users that no item of f_2 matches the new restriction feature f_3.

Theorem 2 (completeness). *Let D be a RDF dataset. The navigation graph G_D is* complete *except for some queries having an unsafe focus change, i.e., for every query q s.t. for every $\phi \in Phi(q)$, $items(q, \phi) \neq \emptyset$, there is a navigation path from v_0 to the navigation place $(\underline{q}_{\phi}, \phi)$.*

In the above scenario, it was possible to navigate to $(f_1 \texttt{ or } f_2) \texttt{ and } f_3$ that has an unsafe focus change on f_2, but it is not possible to navigate to the equivalent $f_3 \texttt{ and } (f_1 \texttt{ or } f_2)$ because $f_2 \notin restr(f_3 \texttt{ and } (f_1 \texttt{ or } \underline{?}_{\phi_2}), \phi_2)$. Fortunately, a query that is not a dead-end but has unsafe focus changes can be simplified into an equivalent query (same set of items) without unsafe focus changes. It suffices to delete from the query empty alternatives ($S \cup \emptyset = S$), and empty exclusions ($S \setminus \emptyset = S$).

4.2 Efficiency

Each navigation step from a navigation place (q, ϕ) requires the computation of the set of items $items(q, \phi)$, the set of restrictions $restr(q, \phi)$, and the set of navigation links as specified in Definition 8. In many cases, the set of items can be obtained efficiently from the previous set of items, and the last navigation link. If the last navigation link was an intersection, Lemma 1 shows that the set of items is the result of the intersection that is performed during the computation of restrictions, like in standard faceted search. For an exclusion or a naming, the set of items is unchanged. For a reference, the set of items was already computed

at the previous step. Otherwise, for an union or a focus change, the set of items is computed with a LISQL query engines, possibly reusing existing query engines for the Semantic Web (see Section 3.3).

Computing the set of restrictions is equivalent to set-based faceted search, i.e., amounts to compute set intersections between the set of items and the precomputed set of items of features. The same datastructures and algorithms can therefore be used. As features are LISQL queries, their set of items can be computed like for queries, possibly with optimizations given features are simple queries. Finally, determining the set of navigation links requires little additional computation. A navigation link is available for each focus of the query (focus change), and each restriction (intersection). Three navigation links for exclusion, union, and naming are always available. Only for reference navigation links it is necessary, for each variable in the query, to compute the set of items of the target navigation place, in order to check it is not empty. This additional cost is limited as the number of variables in a LISQL query is very small in practice, and is bounded by the number of foci of the query.

5 Usability Evaluation

This section reports on the evaluation of QFS in terms of usability[5]. We have measured the ability of users to answer questions of various complexities, as well as their response times. Results are strongly positive and demonstrate that QFS offers expressiveness and ease-of-use at the same time.

Prototype. QFS has been implemented as a prototype, Sewelis[6] (aka. Camelis 2). Figure 1 shows a screenshot of Sewelis. From top to bottom, and from left to right, it is composed of a menu bar (M), a toolbar (T), a query box (Q), query controls (QC), feature controls (FC), an answer list or extension box (E), a facet hierarchy (F), and a set of value boxes (V). A query engine can be derived from Sewelis by retaining only the components Q and E. A standard faceted search system can be derived by retaining only the components E, F, and V. Navigation links, i.e., suggested query transformations, are available on all components. Whenever a navigation control is triggered, the corresponding navigation link is applied, and components (Q,E,F,V) are refreshed accordingly. The query box (Q) is clickable for setting the focus on any subquery. Query controls (QC) provide buttons for *naming*, *union*, *exclusion* (and a few others). Every element of components (E,F,V) can be used as an argument for *intersection*, with the guarantee that the resulting query does have answers. Restriction are dispatched between components (E,F,V) according to their types. The facet hierarchy (F) contains variables of the current query (e.g., `?X`), classes (e.g., `a person`), and property paths (e.g., `father of ?`, `birth : year : ?`). Each value box (V) contains a list or hierarchy of relevant values for some property path facet (e.g.,

[5] Details can be found on `http://www.irisa.fr/LIS/alice.hermann/camelis2.html`

[6] See `http://www.irisa.fr/LIS/softwares/sewelis/` for a presentation, screencasts, a Linux executable, and sample data.

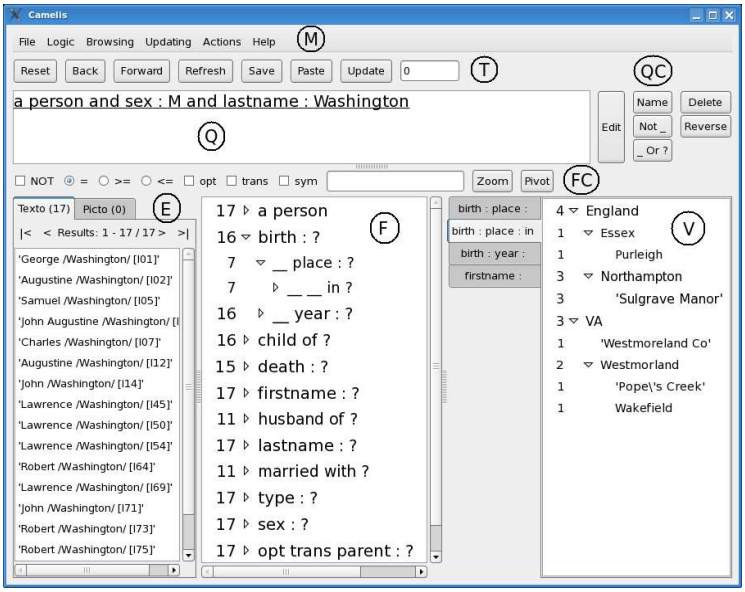

Fig. 1. A screenshot of the user interface of Sewelis. It shows the selection of male persons whose lastname is Washington.

`father of 'George Washington'`, `birth : year : 1601`). The extension box (E) contains resources (e.g., `England`). The hierarchical organization of facets in (F) is based on RDFS class and property hierarchies. A value box (V) is hierarchically organized according to the last property of its property path, if that property is transitive (here, `in` = `part of`).

Dataset. The datasets were chosen so that subjects had some familiarity with the concepts, but not with the individuals. We found genealogical datasets about former US presidents, and converted them from GED to RDF. We used the genealogy of Benjamin Franklin for the training, and the genealogy of George Washington for the test. The latter describes 79 persons by their birth and/or death events, which are themselves described by their year and place, by their firstname, lastname, and sex, and by their relationships (father, mother, child, spouse) to other persons. Places are linked by a transitive *part-of* relationship, allowing for the display of place hierarchies in Sewelis.

Methodology. The subjects consisted of 20 graduate students in computer science. They had prior knowledge of relational databases but neither of Sewelis, nor of faceted search, nor of Semantic Web. None was familiar with the dataset used in the evaluation. The evaluation was conducted in three phases. First, the subjects learned how to use Sewelis through a 20min tutorial, and had 10 more minutes for free use and questions. Second, subjects were asked to answer a set

Category	Question (# navig. links)
Visualization	1 How many persons are there? (0) 2 How many men are there? (0) 3 How many persons have a birth's place in the base? (0)
Selection	4 How many women are named Mary? (4) 5 Who was born at Stone Edge? (4) 6 Which man was born in 1659? (5) 7 Who is married with Edward Dymoke? (3)
Path	9 Which man has his father married with Alice Cooke? (5) 11 Which man is married with a woman born in 1708? (7)
Disjunction	8 Which women have for mother Jane Butler or Mary Ball? (6) 12 Which men are married with a woman whose birth's place is Cuckfields or Stone Edge? (9)
Negation	10 How many men were born in the 1600 or 1700 years, and not in Norfolk? (12) 13 How many women have a mother whose death's place is not Warner Hall? (7)
Inverse	14 Who was born in the same place as Robert Washington? (6) 15 Who died during the year when Augustine Warner was born? (6)
Cycle	16 Which persons died in the same area where they were born? (9) 17 How many persons have the same firstname as one of their parent? (8) 18 Which persons were born the same year as their spouse? (10)

Table 1. Questions of the test, by category, and the minimum number of navigation links to answer them.

of questions, using Sewelis. We recorded their answers, the queries they built, and the time they spent on each question. Finally, we got feedback from subjects through a SUS questionnaire and open questions [1]. The test was composed of 18 questions, with smoothly increasing difficulty. Table 1 groups the questions in 7 categories: the first 2 categories are covered by standard faceted search, while the 5 other categories are not in general. For category *Visualization*, the exploration of the facet hierarchy was sufficient. In category *Selection*, we asked to count or list items that have a particular feature. In category *Path*, subjects had to follow a path of properties. Category *Disjunction* required the use of unions. Category *Negation* required the use of exclusions. Category *Inverse* required the crossing of the inverse of properties. Category *Cycle* required the use of co-reference variables (naming and reference navigation links).

Results. Figure 2 shows the number of correct queries and answers, the average time spent on each question and the number of participants who had a correct query for at least one question of each category. For example, in category "Visualization", the first two questions had 20 correct answers and queries; the third question had 10 correct answers and 13 correct queries; all the 20 participants had a correct query for at least one question of the category; the average re-

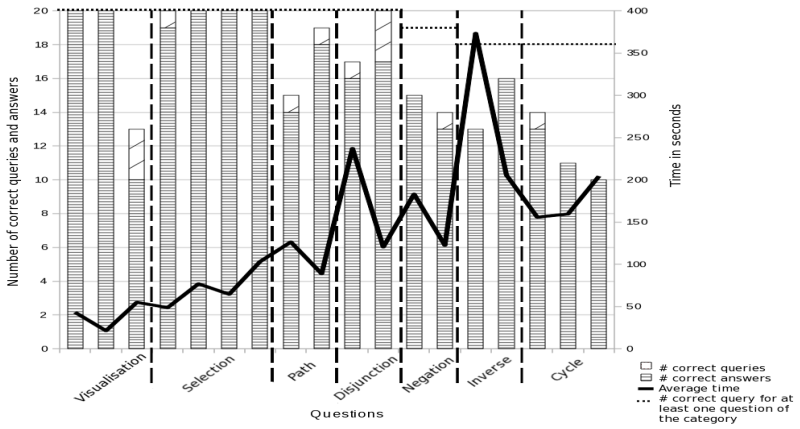

Fig. 2. Average time and number of correct queries and answers for each question

sponse times were respectively 43, 21, and 55 seconds. The difference between the number of correct queries and correct answers is explained by the fact that some subjects forgot to set the focus on the whole query after building the query.

All subjects but one had correct answers to more than half of the questions. Half of the subjects had the correct answers to at least 15 questions out of 18. Two subjects answered correctly to 17 questions, their unique error was on a disjunction question for one and on a negation question for the other. All subjects had the correct query for at least 11 questions. For each question, there is at least 50 percent of success. The subjects spent an average time of 40 minutes on the test, the quickest one spent 21 minutes and the slowest one 58 minutes.

The first 2 categories corresponding to standard faceted search, visualization and selection, had a high success rate (between 94 and 100) except for the third question. The most likely explanation for the latter is that the previous question was so simple (`a man`) that subjects forgot to reset the query between the questions 2 and 3. All questions of the first two categories were answered in less than 1 minute and 43 seconds on average. Those results indicate that the more complex user interface of QFS does not entail a loss of usability compared to standard faceted search for the same tasks.

For other categories, all subjects but two managed to answer correctly at least one question of each category. Within each category, we observed that response times decreased, except for the *Cycle* category. At the same time, for *Path*, *Disjunction* and *Inverse*, the number of correct answers and queries increased. Those results suggest a quick learning process of the subjects. The decrease in category *Negation* is explained by a design flaw in the interface. For category *Cycle*, we conjecture some lassitude at the end of the test. Nevertheless, all but two subjects answered correctly to at least one of *Cycle* questions. The peak of response time in category *Inverse* is explained by the lack of inverse property examples in the tutorial. It is noticeable that subjects, nevertheless, managed

SUS Question	Score (on a 0-4 scale)	
I think that I would like to use this system frequently	2.8	Agree
I found the system unnecessarily complex	0.8	Strongly disagree
I thought the system was easy to use	2.6	Agree
I think that I would need the support of a technical person to be able to use this system	1.5	Disagree
I found the various functions in this system were well integrated	2.9	Agree
I thought there was too much inconsistency in this system	0.6	Strongly disagree
I would imagine that most people would learn to use this system very quickly	2.5	Agree
I found the system very cumbersome to use	1.0	Disagree
I felt very confident using the system	2.8	Agree
I needed to learn a lot of things before I could get going with this system	1.7	Neutral

Table 2. Results of SUS questions.

to solve the *Inverse* questions with a reasonable success rate, and a decreasing response time.

SUS Questionnaire. Table 2 shows the answers to the SUS questions, which are quite positive. The first noticeable thing is that, despite the relative complexity of the user interface, subjects do not find the system *unnecessarily complex* nor *cumbersome to use*. We think this is because the principles of QFS are very regular, i.e., they follow few rules with no exception. The second noticeable thing, which may be a consequence of the first, is that subjects *felt confident using the system* and found no *inconsistency*. Finally, even if it is necessary for subjects to learn how to use the system, they *thought that the system was easy to use*, and that they *would learn to use it very quickly*. The results of the test demonstrate that they are right, even for features that were not presented in the tutorial (the Inverse category).

6 Conclusion

We have introduced *Query-based Faceted Search* (QFS) as a search paradigm for Semantic Web knowledge bases, in particular RDF datasets. It combines most of the expressiveness of the SPARQL query language, and the benefits of exploratory search and faceted search. The user interface of QFS includes the user interface of other faceted search systems, and can be used as such. It adds a query box to tell users where they are in their search, and to allow them to change the focus. It also adds a few controls for applying some query transformations such as the insertion of disjunction, negation, and variables.

QFS has been implemented as a prototype, Sewelis. Its usability has been demonstrated through a user study, where, after a short training, all subjects

were able to answer simple questions, and most of them were able to answer complex questions involving disjunction, negation, or co-references. This means QFS retains the ease-of-use of other faceted search systems, and gets close to the expressiveness of query languages such as SPARQL.

Acknowledgments. We would like to thank the 20 students, from the University of Rennes 1 and the INSA engineering school, for their volunteer participation to the usability evaluation.

References

1. Brooke, J.: SUS: A quick and dirty usability scale. In: Jordan, P., Thomas, B., Weerdmeester, B., McClelland, A. (eds.) Usability evaluation in industry, pp. 189–194. London: Taylor and Francis (1996)
2. Ferré, S.: Camelis: a logical information system to organize and browse a collection of documents. Int. J. General Systems 38(4) (2009)
3. Ferré, S., Hermann, A., Ducassé, M.: Semantic faceted search: Safe and expressive navigation in RDF graphs. Research report, IRISA (2011), http://hal.inria.fr/inria-00410959/PDF/PI-1964.pdf
4. Ferré, S., Ridoux, O.: A file system based on concept analysis. In: Sagiv, Y. (ed.) Int. Conf. Rules and Objects in Databases. pp. 1033–1047. LNCS 1861, Springer (2000)
5. Hearst, M., Elliott, A., English, J., Sinha, R., Swearingen, K., Yee, K.P.: Finding the flow in web site search. Communications of the ACM 45(9), 42–49 (2002)
6. Heim, P., Ertl, T., Ziegler, J.: Facet graphs: Complex semantic querying made easy. In: et al., L.A. (ed.) Extended Semantic Web Conference. pp. 288–302. LNCS 6088, Springer (2010)
7. Hildebrand, M., van Ossenbruggen, J., Hardman, L.: /facet: A browser for heterogeneous semantic web repositories. In: *et al*, I.C. (ed.) Int. Semantic Web Conf. pp. 272–285. LNCS 4273, Springer (2006)
8. Kaufmann, E., Bernstein, A.: Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. J. Web Semantics 8(4), 377–393 (2010)
9. Lu, J., Ma, L., Zhang, L., Brunner, J., Wang, C., Pan, Y., Yu, Y.: SOR: A practical system for ontology storage, reasoning and search (demo). In: Int. Conf. Very Large Databases (VLDB). pp. 1402–1405. VLDB Endowment, ACM (2007)
10. Marchionini, G.: Exploratory search: from finding to understanding. Communications of the ACM 49(4), 41–46 (2006)
11. Oren, E., Delbru, R., Decker, S.: Extending faceted navigation to RDF data. In: *et al*, I.C. (ed.) Int. Semantic Web Conf. pp. 559–572. LNCS 4273, Springer (2006)
12. Sacco, G.M.: Dynamic taxonomies: A model for large information bases. IEEE Transactions Knowledge and Data Engineering 12(3), 468–479 (2000)
13. Sacco, G.M., Tzitzikas, Y. (eds.): Dynamic taxonomies and faceted search. The information retrieval series, Springer (2009)
14. Tran, T., Wang, H., Haase, P.: Hermes: Data web search on a pay-as-you-go integration infrastructure. Web semantics: Science, Services and Agents on the World Wide Web 7, 189–203 (2009)